BME646 and ECE60146: Homework 3

Spring 2026
Due Date: Tuesday, Feb 3, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. submissions will be accepted with penalty: -10 points per-late-day, up to 5 days.

1 Introduction

In this homework you will develop an understanding of advanced momentum-based optimization techniques in deep neural network training by implementing three modern optimizers: Nesterov Accelerated Gradient (NAG), RMSprop, and AdamW. You will also perform hyperparameter tuning to understand how weight decay and momentum affect optimizer performance.

1.1 Context and Motivation

As discussed in Prof. Kak's lecture on Autograd [1], vanilla Stochastic Gradient Descent (SGD) suffers from several problems:

- Slow convergence, especially near the optimum (Slide 20)
- Oscillations when the loss surface forms a narrow valley (Slides 107-108)
- Inability to adapt learning rates to different parameters with sparse gradients (Slides 112-113)

This homework addresses these problems through a progression of increasingly sophisticated optimizers:

1. Nesterov Accelerated Gradient (NAG): Extends the momentum concept from Slides 109-111 by adding a "lookahead" mechanism that evaluates gradients at predicted future positions rather than current positions. This provides better convergence properties, especially in narrow valleys.

2. RMSprop: Builds on the AdaGrad algorithm (Slides 114-115) and addresses its vanishing learning rate problem (Slide 116) by using a moving average rather than accumulating all past gradients. This allows the

optimizer to adapt learning rates per-parameter while maintaining learning throughout training.

3. AdamW: Extends the Adam optimizer (Slides 117-118) by decoupling weight decay from the gradient-based update. This has become the default optimizer in modern deep learning frameworks and achieves superior generalization.

For complete background on the concepts covered in this homework, please refer to Prof. Kak's lecture notes on Autograd [1], particularly Sections 2, 3, and 11.

1.2 Quick Reference Guide to Lecture Slides

For your convenience, here are the most relevant sections from the lecture notes:

- **Slides 19-20:** Problems with vanilla Gradient Descent
- **Slides 21-31:** Stochastic Gradient Descent (SGD) fundamentals
- **Slides 52-62:** One-neuron classifier implementation (basis for homework)
- **Slides 63-82:** Multi-neuron classifier implementation (basis for homework)
- **Slides 107-111:** Momentum concepts (foundation for NAG)
- **Slides 112-116:** Sparse gradients and AdaGrad/RMSprop
- **Slides 117-118:** Adam optimizer (foundation for AdamW)

2 Using ComputationalGraphPrimer

1. Download the `tar.gz` archive and install version 1.1.4 of your instructor's ComputationalGraphPrimer. Do NOT `sudo pip install` the Primer since that would not give you the Examples directory of the distribution that you are going to need for the homework. The main documentation page for the Primer can be accessed though the following link:

   https://engineering.purdue.edu/kak/distCGP/

2. Execute the following scripts in the Examples directory:

   `python one_neuron_classifier.py`

   `python multi_neuron_classifier.py`

   The final output of both these scripts is a display of the training loss versus the training iterations.

3. Now, execute the following script in the Examples directory

   `python verify_with_torchnn.py`

   If you did not make changes to the script in the Examples directory, the loss vs. iterations graph that you will see is for a network that is a `torch.nn` version of the handcrafted network you get through the script `multi_neuron_classifier.py`

   Compare visually the output you get with the above call with what you saw for the second script in Step 2.

4. Now make appropriate changes to the file `verify_with_torchnn.py` in order to see the `torch.nn` based output for the one-neuron model. The changes you need to make are mentioned in the documentation part of the file `verify_with_torchnn.py`.

   Again compare visually the loss-vs-iterations for the one-neuron case with the handcrafted network vis-a-vis the `torch.nn` based network.

5. Now comes the main part of this homework:

   If you would look at the code for the one-neuron and multi-neuron models in the Primer, you will notice that the step-size calculations do not use any optimizations. [For the one-neuron case, you can also see the backprop and update code on Slide 59 and, for the multi-neuron case, on Slide 80 of the Week 3 slides.] The implemented parameter update steps are based solely on the current value of the gradient of the loss with respect to the parameter in question. That is,

   $$p_{t+1} = p_t - \text{lr} \cdot g_{t+1} \tag{1}$$

   where p_t denotes learnable parameters from the previous time step, *e.g.* layer weights at iteration t, and g_{t+1} denotes the corresponding gradient for the current time step $t + 1$.

   It is your job to improve the estimation of p_{t+1} using modern momentum-based optimization techniques. In order to fully appreciate what that

means, it is recommended that you carefully review the material on Slides 106 through 118 of the Week 3 slides[1].

What follows is a detailed description of the three optimizers you will implement in order to complete your homework.

- **Nesterov Accelerated Gradient (NAG)**:
  Motivation (from Slides 109-111): Classical momentum, as introduced in the lecture, computes the step update using:

$$\begin{aligned} \delta p_t &= \gamma \cdot \delta p_{t-1} + 2 \cdot \alpha \cdot J_F(p_t) \cdot \epsilon_t \\ p_{t+1} &= p_t - \delta p_t \end{aligned} \tag{2}$$

  While momentum helps with oscillations (Slides 107-108), it has a limitation: it evaluates the gradient at the current position p_t before adding momentum. Nesterov's key insight is to evaluate the gradient at the *lookahead position* where momentum would take us.

  Nesterov Momentum (NAG) improves upon classical momentum by providing better convergence guarantees, especially in narrow valleys. The key difference is the "lookahead" strategy: instead of computing the gradient at the current position and then adding momentum, NAG first makes a jump using the momentum term, then computes the gradient at that new position.

  The update equations are:

$$\begin{aligned} v_{t+1} &= \mu \cdot v_t + g(p_t - \mu \cdot v_t), \\ p_{t+1} &= p_t - \mathrm{lr} \cdot v_{t+1}. \end{aligned} \tag{3}$$

  Here, $g(p_t - \mu \cdot v_t)$ represents the gradient evaluated at the lookahead position. The term $(p_t - \mu \cdot v_t)$ is where pure momentum would take us, and we evaluate the gradient there before making the final update. For simplicity, constant scaling factors are absorbed into the learning rate.

  Implementation steps:

  (a) Compute the lookahead position: $p_{\text{lookahead}} = p_t - \mu \cdot v_t$
  (b) Evaluate the gradient at this lookahead position
  (c) Update velocity: $v_{t+1} = \mu \cdot v_t + g(p_{\text{lookahead}})$
  (d) Update parameters: $p_{t+1} = p_t - \mathrm{lr} \cdot v_{t+1}$

The momentum hyperparameter $\mu \in [0, 1]$ controls the strength of the lookahead. Typical values range from 0.9 to 0.99. The velocity v_0 is initialized to zero.

Connection to lecture: NAG directly addresses the oscillation problem discussed in Slides 107-109 by "peeking ahead" before committing to a step, resulting in a smoother, more stable solution path.

You can find the PyTorch documentation for NAG (via SGD with nesterov=True) at:

https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

For the original theory, see Nesterov's 1983 paper: *"A method for solving the convex programming problem with convergence rate O(1/k²)"*

- **RMSprop (Root Mean Square Propagation)**:

  Motivation (from Slides 112-116): The lecture discusses the problem of *sparse gradients*, when different training samples are sensitive to different subsets of parameters. This is common in neural networks where different features activate for different inputs.

  AdaGrad (Slides 114-115) addressed sparse gradients by adapting the learning rate per-parameter:

$$p_{t+1}^i = p_t^i - \frac{\alpha}{\sqrt{\sum_t (g_t^i)^2 + \epsilon}} \cdot g_t^i \tag{4}$$

  However, as noted in Slide 116, AdaGrad's monotonically increasing denominator causes learning rates to vanish over time, stopping learning prematurely.

  RMSprop fixes this by replacing the cumulative sum with a *moving average* of squared gradients, allowing the optimizer to "forget" old gradients and continue learning throughout training.

  The update equations are:

$$\begin{aligned} v_{t+1} &= \beta \cdot v_t + (1 - \beta) \cdot (g_{t+1})^2, \\ p_{t+1} &= p_t - \text{lr} \cdot \frac{g_{t+1}}{\sqrt{v_{t+1}} + \epsilon}. \end{aligned} \tag{5}$$

  In these equations:

- v_t is the moving average of squared gradients (initialized to zero)
- $\beta \in [0, 1]$ controls the decay rate (typically 0.9 or 0.99)
- ϵ is a small constant (typically 10^{-8}) for numerical stability
- The division by $\sqrt{v_{t+1}}$ adapts the learning rate per-parameter
- Note that $(g_{t+1})^2$ means element-wise squaring of the gradient vector

Why moving average? Unlike AdaGrad's cumulative sum, the moving average v_t can decrease if recent gradients are smaller, allowing learning rates to recover. The decay factor β controls how much history to remember: larger β means more history, smaller β means faster adaptation.

RMSprop is particularly effective for non-stationary objectives and is widely used in recurrent neural networks. Note that RMSprop uses only the second moment (squared gradients) unlike Adam which uses both first and second moments.

You can find the PyTorch documentation for RMSprop at:

https://pytorch.org/docs/stable/generated/torch.optim.RMSprop.html

Additional lecture slides on RMSprop at:

https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

- **AdamW (Adam with Decoupled Weight Decay)**:

  Motivation (from Slides 117-118): The lecture introduces Adam as combining momentum (first moment) with adaptive learning rates (second moment). As stated in Slide 117:

  "The notion of momentum boils down to keeping a running decaying average of the mean of the past gradient componentsthis would be the first moment of the gradient components. And the need to be adaptive to the different components of the gradient as in AdaGrad/RMSprop requires that we keep running tabs on the second moment of the gradient component values."

  Adam is one of the most widely used optimizers in deep learning. However, the original Adam implementation had an issue with how it handled L2 regularization (weight decay). AdamW fixes this by *decoupling* weight decay from the gradient-based update, leading to better generalization.

The standard Adam update (from Slides 117-118) is:

$$\begin{aligned} m_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1) \cdot g_{t+1}, \\ v_{t+1} &= \beta_2 \cdot v_t + (1-\beta_2) \cdot (g_{t+1})^2, \\ \hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}}, \\ \hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}}, \\ p_{t+1} &= p_t - \mathrm{lr} \cdot \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon}. \end{aligned} \tag{6}$$

AdamW modifies the final update step to include decoupled weight decay:

$$p_{t+1} = p_t - \mathrm{lr} \cdot \left(\frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon} + \lambda \cdot p_t \right), \tag{7}$$

where λ is the weight decay coefficient (typically 1e-4 to 1e-2 (occasionally higher for large models)).

Key difference from Adam: The critical change is the $+\lambda \cdot p_t$ term applied *directly to the parameters*, separate from the adaptive learning rate computation. In regular Adam with L2 regularization, weight decay would be added to the gradient *before* adaptive scaling, which interacts poorly with the adaptive learning rates.

Understanding the components:

- m_t: First moment (momentum), running average of gradients (as in Slide 117)
- v_t: Second moment, running average of squared gradients (for adaptation)
- β_1 and β_2: Decay rates for first and second moments (typically 0.9 and 0.999)
- $\hat{m}_t$ and $\hat{v}_t$: Bias-corrected moments (see Slide 118, needed because $m_0 = v_0 = 0$)
- λ: Weight decay strength (regularization parameter)
- ϵ: Numerical stability constant (typically 10^{-8})

Why bias correction? As noted in Slide 118, initializing $m_0 = v_0 = 0$ biases estimates toward zero in early iterations. The correction terms $\frac{1}{1-\beta_1^{t+1}}$ and $\frac{1}{1-\beta_2^{t+1}}$ compensate for this, especially important in the first few training steps.

AdamW is the default optimizer in many modern frameworks and is particularly effective for training large models like transformers.

You can find the PyTorch documentation for AdamW at:

https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html

The AdamW algorithm was introduced by Loshchilov & Hutter in their 2019 ICLR paper: *"Decoupled Weight Decay Regularization"*. You can access the paper at:

https://arxiv.org/abs/1711.05101

For the original Adam paper (Kingma & Ba, 2015), see:

https://arxiv.org/abs/1412.6980

- **Hyperparameter Tuning for Modern Optimizers**:

  Hyperparameter tuning is crucial in deep learning as it involves optimizing the settings that control the learning process, directly impacting model performance. The right hyperparameter values can significantly enhance a model's accuracy, convergence speed, and generalization ability.

  For this homework, you will explore:

  - **Momentum (μ) for NAG**: How does the lookahead strength affect convergence?
  - **Weight decay (λ) for AdamW**: How does regularization strength affect the final loss?

  This exercise will provide insights into how modern optimizers balance fast convergence with good generalization through their hyperparameters.

3 Programming Tasks (100 Points Total)

1. **Implementing NAG, RMSprop, and AdamW (60 points):** Modify the existing one-neuron and multi-neuron classifiers to incorporate modern momentum-based optimization techniques. Implement three updated versions of these classifiers:

   - **Nesterov Accelerated Gradient (NAG) - 20 points:** Incorporates lookahead momentum for better convergence
   - **RMSprop - 20 points:** Adapts learning rates per-parameter based on gradient history

- **AdamW - 20 points:** Combines adaptive learning rates with proper weight decay

Grading criteria for each optimizer:

- Correct implementation of update equations (10 points)
- Proper initialization of optimizer variables (3 points)
- Working code that trains successfully (5 points)
- Code quality and documentation (2 points)

NOTE: Preserve Original Code Structure
Do not modify the primary module file `ComputationalGraphPrimer.py`. Instead, create subclasses that inherit from `ComputationalGraphPrimer` and implement or override methods as necessary.

2. **Performance Analysis (15 points):** Compare the results of vanilla SGD, NAG, RMSprop, and AdamW in terms of:

   - Convergence speed (how quickly loss decreases) - 3 points
   - Final loss values achieved - 3 points
   - Stability during training (smoothness of loss curves) - 3 points

   Provide your observations and explain (6 points):

   - Why NAG might converge faster than vanilla momentum
   - How RMSprop's adaptive learning rates help with convergence
   - Why AdamW often achieves the best final loss values

3. **Hyperparameter Sensitivity Analysis (15 points):** Explore how optimizer performance depends on key hyperparameters:

   - For **NAG**: Train with three different momentum values μ (e.g., [0.8, 0.9, 0.99]) - 5 points
   - For **AdamW**: Train with three different weight decay values λ (e.g., [0.001, 0.01, 0.1]) - 5 points

   Create a table showing for each configuration (5 points):

   - Training time (in seconds or iterations)
   - Final loss value
   - Minimum loss achieved during training

- Any observations about training stability

4. **Visualizing Results (10 points):** Generate comparative plots to illustrate the improvements:

   - Loss vs. iterations for all optimizers (vanilla SGD, NAG, RMSprop, AdamW) on the same plot - 5 points
   - Separate plots showing the effect of different hyperparameter values for NAG and AdamW - 3 points
   - Clear labels, legends, and proper formatting - 2 points

   Note: There is no single "correct" result. Your outcomes may vary depending on training parameters such as learning rate, batch size, and number of iterations. Focus on understanding the *relative* performance differences between optimizers.

4 Submission Instructions

Include a typed report explaining how you solved the given programming tasks. You may refer to the homework solutions posted at the class website for previous years for examples of how to structure your report.

1. **Turn in a PDF file and mark all pages on Gradescope**.
2. Submit your code file(s) as a zip file.
3. **Code and Output Placement:** Include the output directly next to the corresponding code block in your submission. Avoid placing the code and output in separate sections as this can make it difficult to follow.
4. **Output Requirement:** Ensure that all your code produces outputs and that these outputs are included in the submitted PDF. Submissions without outputs may not receive full credit, even if the code appears correct.
5. For this homework, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert code to `.py` and submit that as source code. **Do NOT submit .ipynb notebooks.**

6. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

7. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

5 Frequently Asked Questions (FAQ)

Q: Where exactly should I implement the optimizers?

A: Do NOT modify `ComputationalGraphPrimer.py` directly. Instead, create a new Python file (e.g., `my_optimizers.py`) that imports CGP and defines subclasses. For example:

```
from ComputationalGraphPrimer import *

class MyCustomCGP(ComputationalGraphPrimer):
    def __init__(self, optimizer_type='nag', **kwargs):
        super().__init__(**kwargs)
        self.optimizer_type = optimizer_type
        # Initialize optimizer-specific variables here

    # Override the backprop functions here
```

Q: How do I handle the lookahead step in NAG?

A: The lookahead requires temporarily modifying parameters. Here is the key insight:

1. Save current parameter values
2. Compute lookahead position: $p\_lookahead = p - \mu \cdot v$
3. Set parameters to lookahead position
4. Compute gradients at this position
5. Restore original parameters
6. Update using the lookahead gradients

Be sure to restore original parameters after computing lookahead gradients, otherwise training will silently break.

Q: What does "element-wise" mean in the equations?

A: Element-wise operations apply to each element independently. For example, if $g = [2, -1, 3]$ and we compute g^2 element-wise, we get $[4, 1, 9]$, NOT a dot product. In Python/NumPy: `g**2` or `np.square(g)`.

Q: How do I initialize the velocity and moment variables?

A: All velocity (v), momentum (m), and second moment variables should be initialized to zero with the same shape as your parameters. For example:

```
self.velocity = {param: np.zeros_like(param) for param in self.
    learnable_params}
self.m = {param: np.zeros_like(param) for param in self.
    learnable_params} # for Adam
self.v = {param: np.zeros_like(param) for param in self.
    learnable_params} # for Adam
```

Q: The bias correction in Adam, when do I apply it?

A: Apply bias correction on EVERY iteration. The correction terms $\frac{1}{1-\beta_1^t}$ and $\frac{1}{1-\beta_2^t}$ are computed fresh each iteration using the iteration number t. Do not forget that t starts from 1, not 0.

Q: What is the actual difference between Adam and AdamW?

A: A single line of code. In Adam, if you want weight decay, you would add $\lambda \cdot p$ to the gradient BEFORE the adaptive scaling. In AdamW, you add it to the parameter update AFTER adaptive scaling:

```
# Adam with L2 regularization (WRONG way for weight decay)
g = g + lambda * p  # added to gradient
p = p - lr * m_hat / (sqrt(v_hat) + eps)

# AdamW (CORRECT way for weight decay)
p = p - lr * (m_hat / (sqrt(v_hat) + eps) + lambda * p)
```

Q: Why does RMSprop use $\sqrt{v_t}$ instead of just v_t?

A: This keeps the adaptive term in the same units as the gradient. Since v_t is the average of *squared* gradients, taking the square root gives us something with the same units as the gradient itself, making the division meaningful.

Q: Can momentum be greater than 1?

A: No. Momentum coefficients (μ, β_1, β_2) must be in $[0, 1]$. Values greater than 1 would cause exponential growth and instability. Typical values are 0.9-0.99.

Q: My loss is NaN or exploding. What is wrong?

A: Common causes:

- Learning rate too high (try reducing by 10x)
- Forgot to initialize optimizer variables to zero

- Division by zero: make sure you add ϵ in denominators
- Forgot bias correction in Adam (early iterations can have huge updates)

Q: My optimizer performs worse than vanilla SGD. Why?
A: This can happen. Possible reasons:

- Learning rate not tuned for the optimizer (each optimizer needs different lr)
- Hyperparameters not set correctly (e.g., β_1, β_2, λ)
- Bug in implementation (check against PyTorch's version)
- For some simple problems, vanilla SGD can be sufficient

Q: How do I know if my implementation is correct?
A: Three validation methods:

1. Compare loss curves with `verify_with_torchnn.py`
2. Check that loss decreases monotonically (mostly)
3. Print intermediate values and verify they match equations
4. Test on the same data with PyTorch's built-in optimizer

Q: What learning rate should I use?
A: It depends on the optimizer:

- NAG: Similar to vanilla SGD, try 1e-3 to 1e-1
- RMSprop: Usually smaller, try 1e-3 to 1e-2
- AdamW: Often works well with 1e-3 to 1e-4

When switching optimizers, always re-tune the learning rate.

Q: How do I choose momentum values for tuning?
A: Use a logarithmic or geometric spread:

- Low momentum: 0.8 (fast adaptation, less smoothing)
- Medium momentum: 0.9 (balanced, often default)
- High momentum: 0.99 (heavy smoothing, slower to change)

Q: What about weight decay values?
A: Common ranges:

- Very light: 1e-4 to 1e-3
- Light: 1e-3 to 1e-2 (typical default)
- Moderate: 1e-2 to 5e-2
- Strong: 5e-2 to 1e-1

Stronger weight decay helps generalization but may hurt training loss.

Q: How many training iterations should I run?
A: For the homework examples, 20,000-40,000 iterations should be sufficient to see clear trends. If your loss has not started decreasing after 1,000 iterations, something is likely wrong.

Q: Should I use the same batch size for all optimizers?
A: Yes, keep batch size constant (e.g., 8 as in the examples) when comparing optimizers. Changing batch size affects convergence and makes fair comparison difficult.

Q: How do I make good comparison plots?
A: Tips for clear visualizations:

- Plot all optimizers on the same graph with different colors
- Use a legend to identify each curve
- Label axes clearly ("Training Iteration" and "Loss")
- Consider using log scale for y-axis if loss varies over orders of magnitude
- Smooth noisy curves with a moving average window

Q: My code runs very slowly. Is this normal?
A: The handcrafted networks will be slower than PyTorch's optimized versions, but should not be prohibitively slow. If training 40,000 iterations takes more than a few minutes, you might have inefficient loops. Make sure you are not recomputing things unnecessarily.

Q: How detailed should my analysis be?
A: Your report should include:

- Brief explanation of what you implemented (1-2 paragraphs per optimizer)

- Clear presentation of results (plots and tables)
- Thoughtful analysis comparing the optimizers
- Observations from hyperparameter tuning

You do not need to explain the mathematical derivations (we covered those in lecture), but do explain your implementation approach and results.

Q: Should I include code in my report?

A: Include key code snippets (10-20 lines) showing the core update logic for each optimizer. Do not paste your entire codebase. Show the most important parts and explain what they do. Your entire code should only be submitted as a zip file to the code submission link for HW3.

References

[1] Autograd Lecture. URL https://engineering.purdue.edu/kak/pdf-kak/AutogradAndCGP.pdf.

[2] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning, lecture 6e - rmsprop: Divide the gradient by a running average of its recent magnitude. Coursera, 2012. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL https://arxiv.org/abs/1412.6980.

[4] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1711.05101.

[5] Yurii Nesterov. A method for solving the convex programming problem with convergence rate o($1/k^2$). *Soviet Mathematics Doklady*, 27(2):372–376, 1983.